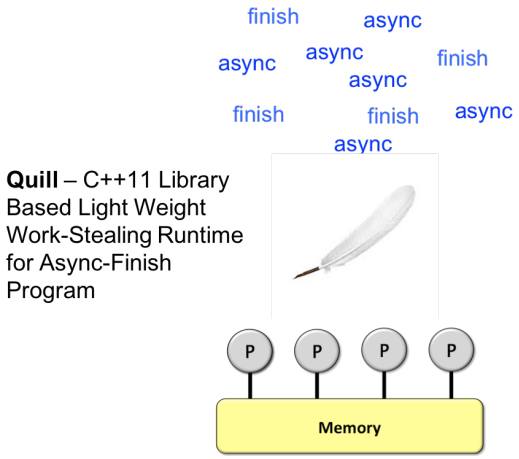
Homework Assignment-2

Due by 11:59pm on 24th February 2023

(Total 12% weightage)

Instructor: Vivek Kumar

QUILL: A light-weight work-stealing runtime for async-finish parallelism



No extensions will be provided. Any submission after the deadline will not be evaluated. If you see an ambiguity or inconsistency in a question, please seek a clarification from the teaching staff.

Plagiarism: **This is an individual assignment.** All submitted programs are expected to be the result of your individual effort. You should never misrepresent someone else's work as your own. In case any plagiarism is detected, it will be dealt **strictly** as per the new plagiarism policy of IIITD.

Please also note that discussing the approach/solution of assignment with your batch mate, i.e. using your friend's idea is also plagiarism.

Instructions

1) Hardware requirements:
   a. You will require a Linux/Mac OS to complete this assignment.
   b. You can easily code/complete this assignment on your laptop.
2) Software requirements are:
   a. Pthread library.
   b. C++11 compiler.
   c. GNU make.
   d. You **must** do version controlling of all your code using github. You should only inside a **PRIVATE** repository. If you are found to be using a **PUBLIC** access repository then it will be considered as plagiarism. NOTE that TAs will check your github repository during demo.
3) **Reference materials** – This assignment will be based on concepts taught in Lectures 02, 07, and 08. Apart from that you will also require some of the code that you wrote in

assignment-1 (StaMp). You are allowed to reuse that code in this assignment. You will also require use of pthread-specific data key creation that has not been covered in lectures but you can easily understand that by going through any online documentation [1].

Learning Objectives

1) Hands-on over C and C++11
2) Learning C++11 lambda functions declaration and usage
3) Learning Pthread APIs (thread creation, termination, synchronization and thread-specific data key creation)
4) Learning Makefiles
5) Able to understand the importance of productivity and performance in parallel programming

Assignment

In this assignment you have to implement a **library-based** new concurrency platform called as "***quill***". As the name suggests, quill will be a very lightweight implementation that should support basic async-finish based task-parallelism. The requirements for quill are as following:

Linguistic Interfaces

1) The linguistic interfaces for exposing the parallelism are "async", "start_finish", and "end_finish". Signatures of these APIs are as following:

```
void async(std::function<void()> &&lambda); //accepts a C++11 lambda function
void start_finish();
void end_finish();
```

2) Pseudo-code of a parallel program that can be executed using quill is as shown below:

```
#include "quill.h"                    // quill APIs are declared here
int main(int argc, char** argv) {
   quill::init_runtime();             // initialize quill runtime
   quill::start_finish();             // start a "flat-finish" scope
      quill::async (S1);              // spawn an async
      S2;
   quill::end_finish();               // end the "flat-finish" scope
   quill::finalize_runtime();         // finalize and release runtime resources
   return 0;
}
```

Quill should support very basic async-finish parallelism. It should be able to support "**flat finish**" scopes. A flat-finish scope is a single-level finish scope for all async tasks spawned within its scope (i.e., an async task can recursively create new async tasks but it can never spawn new finish scopes).

Runtime Implementation

Quill runtime should be a library-based **thread pool** based implementation that would rely on "**work-stealing**" technique for task scheduling. The rough code sketch for this type of runtime implementation was explained in Lecture 07 and Lecture 08. You are encouraged to use that code but remember that was pseudo-code and for actual implementation you will have to fill up the missing parts.

Requirements:

a) You should implement quill in C++ but should not use std::thread / mutex.
b) Proper documentation should be done in your coding.
c) All runtime APIs should be within the "quill" namespace (see pseudo-code above).
d) As explained in Lecture 08, your work-stealing runtime should have per-thread deque datastructure. **This deque must be an array datastructure. Deque from C++ STL is not allowed.** To access this deque you will need task counters at both the ends of the deque. The bottom (or tail) will be private to victim (owner) and will be used in LIFO order. The top (or head) will only be accessed by a thief during steal operation (FIFO order).
e) To implement concurrent deque accesses you should **only** use pthread_mutex_lock/pthread_mutex_unlock APIs. You won't be able to implement the lockless deque explained in lectures without using memory barriers and atomic compare-and-swap operations that are not covered in lectures. Hence, for your deque implementation, push() operation will always be lockless whereas steal() and pop() would always be synchronized.
f) Each Quill worker should be assigned a unique integer id in range 0 to N-1 (N= quill workers) at the time of thread creation. You would need this id during deque operations. You must retrieve this id by using pthread-specific data key creation APIs [1].
g) The size of the deque should be a compile time constant.
h) Your quill runtime should throw proper error if deque becomes full. In such scenarios, increasing the deque size (a compile time constant as explained above), rebuilding quill runtime and your testcase should resolve this issue.
i) Quill users should be able to control quill's thread pool size by using the environment variable "QUILL_WORKERS" (e.g. usage: export QUILL_WORKERS=4). Default thread pool size should be 1. The master thread should also be a part of the thread pool.

Your assignment directory should compose of following files **ONLY** (don't change the file names and its functionality):

a. **quill.h**: This header file should only contain the declarations of async, start_finish, end_finish, init_runtime, and finalize_runtime (see the pseudo-code for input program above).
b. **quill-runtime.cpp**: This should contain all of your quill runtime implementation.
c. **quill-runtime.h:** This header file should contain the declarations of all the routines you implemented inside quill-runtime.cpp.
d. **Makefile:** This should be able to compile your implementation of quill as well as the standard testcase that we are providing herewith (see below).

We will evaluate your implementation of quill using the NQueens application (with input **N=12**). A parallel version of NQueens application is provided herewith that is written using quill APIs. You should not do any changes to this application as for evaluation the instructor will use his own copy of NQueens. This is the same copy that you have been provided along with this assignment. We will consider the correctness of your implementation only if NQueens executes successfully for any value of thread pool size between 1 and **8** (inclusive). Your quill runtime should cleanly exit after NQueens completes. An example of NQueens execution using quill is as following:

```
QUILL_WORKERS=4 ./NQueens 12
```

Your assignment submission should consist of two parts:

a) A tar file containing your source files as mentioned above in the requirements.
b) Your writeup (as PDF inside the above tar) that explains the speedup graph of execution NQueens with threads=1,2,3, and 4; with input as 12.

Reference:

[1] http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_key_create.html